Array Design and Expression Evaluation in POOMA II

Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney,
William Humphrey, John Reynders, Stephen Smith, and Timothy J. Williams

Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM 87545

{karmesin,jac,julianc,swhaney,bfh,reynders,sa_smith,zippy}@lanl.gov

Abstract. POOMA is a templated C++ class library for use in the development of large-scale scientific simulations on serial and parallel computers. POOMA II is a new design and implementation of POOMA intended to add richer capabilities and greater flexibility to the framework. The new design employs a generic `Array` class that acts as an interface to, or *view* on, a wide variety of data representation objects referred to as *engines*. This design separates the interface and the representation of multidimensional arrays. The separation is achieved using compile-time techniques rather than virtual functions, and thus code efficiency is maintained. POOMA II uses PETE, the Portable Expression Template Engine, to efficiently represent complex mathematical expressions involving arrays and other objects. The representation of expressions is kept separate from expression evaluation, allowing the use of multiple evaluator mechanisms that can support nested where-block constructs, hardware-specific optimizations and different run-time environments.

1 Introduction

Scientific software developers have struggled with the need to express mathematical abstractions in an elegant and maintainable way without sacrificing performance. The POOMA (Parallel Object-Oriented Methods and Applications) framework [1, 2], written in ANSI/ISO C++, has demonstrated both high expressiveness and high performance for large-scale scientific applications on platforms ranging from workstations to massively parallel supercomputers. POOMA provides high-level abstractions for multidimensional arrays, physical meshes, mathematical fields, and sets of particles. POOMA also exploits techniques such as expression templates [3] to optimize serial performance while encapsulating the details of parallel communication and supporting block-based data compression. Consequently, scientists can quickly assemble parallel simulation codes by focusing directly on the physical abstractions relevant to the system under study and not the technical difficulties of parallel communication and machine-specific optimization.

POOMA II is a complete rewrite of POOMA intended to further increase expressiveness and performance. The array and field concepts have been redesigned to use a powerful and flexible view-based architecture that decouples interface and representation. Expressions involving arrays and fields are packaged and manipulated using an enhanced version of PETE, the Portable Expression Template Engine. These expressions can operate on subsets of the data, specified via multiple-dimensional domain objects. Finally, the expressions are efficiently evaluated by evaluator objects. These evaluators support a variety of run-time systems, ranging from immediate serial evaluation to thread-based parallel evaluation, as well as complex constructs like where-blocks.

2 Arrays and Engines

An array is a logically rectilinear, N-dimensional table of numeric elements. Most array implementations store their data in a contiguous block of memory and apply Fortran or C conventions for interpreting this data as a multidimensional array. Unfortunately, these two storage conventions do not span the full range of array types encountered in scientific computing: diagonal, banded, symmetric, sparse, etc. One can even imagine arrays that use no storage, computing their element values as functions of their indices or via expressions involving other arrays. One approach to dealing with differing array storage strategies is to simply create new array classes for each case: `BandedArray`, `SparseArray`, and so on. However, this is wasteful since all of these variants have very similar interfaces.

POOMA II's array class provides a uniform interface independent of how the data is stored or computed, without incurring the overhead of C++ virtual function calls. This is accomplished by introducing the concept of an *engine*. An engine is an object that provides a common interface for randomly accessing and changing elements without the need for the user of the engine to know how the elements are stored. For example, an engine that manages a 100×200 "brick" of double-precision values is declared as:

```
Engine<2, double, Brick> brick(100, 200);
```

The domain of this engine is the tensor product of $[0 \ldots 99]$ by $[0 \ldots 199]$. Similarly, an engine that manages a brick of data distributed across a parallel machine in a manner specified by an object `layout` is declared as:

```
Engine<2, double, Distributed> dbrick(100, 200, layout);
```

The domain and range of `dbrick` are identical to that of `brick`, as is the interface for accessing elements. However, the implementations are quite different.

Note that engine classes are all specializations of a common template, `Engine`. A tag is used to specify a particular engine, such as `Brick` or `Distrubuted`, allowing useful default template parameters to be chosen for the array class.

Engines represent a low-level abstraction: getting single elements from a data source. The POOMA II array facility provides an efficient, high-level interface to engines. POOMA II arrays are declared as follows:

```
Array<2, double, Brick>        A(100, 200);
Array<2, double, Distributed>  B(100, 200, layout);
```

This is a variant of the envelope-letter idiom [4]. `Array` (the envelope) delegates all operations to the particular sort of engine (the letter) that it contains. However, compile-time polymorphism, rather than run-time polymorphism, is used for faster performance. In POOMA II, the engines own the data and arrays simply provide an interface for viewing and manipulating that data. In this sense they have semantics similar to iterators in the Standard Template Library [5], except that they automatically dereference themselves. To enforce `const` correctness, POOMA II provides a `ConstArray` class (similar to the STL `const_iterator`) that prohibits modification of its elements.

3 Domains and Views

Domain objects represent the region or set of points on which an array will define values. An N-dimensional domain is composed of N one-dimensional domains and represents the tensor product of these domains. POOMA II includes several domain classes:

1. `Loc<N>`: A single point in N-dimensional space.
2. `Interval<N>`: The tensor product of `N` one-dimensional sequences each having unit stride.
3. `Range<N>`: Similar to `Interval<N>`, with strides specified at run time.
4. `Index<N>`: Similar to `Range<N>`, but with special loop-ordering semantics (see below).
5. `Region<N>`: Tensor product of N one-dimensional continuous domains.

Users choose the domain type that best expresses any constraints that they wish to impose on the domain. For example, `Interval` is used for unit-stride domains and `Loc` is used for single-point domains. This allows POOMA II to infer properties of the domain at compile time and optimize code accordingly.

One of the primary uses of domains is to specify subsections of `Array` objects. Subarrays are a common feature of array classes; however, it is often difficult to make such subarrays behave like first-class objects. The POOMA II engine concept provides a clean solution to this problem: subsetting an `Array` with a domain object creates a new `Array` that has a *view* engine. For example:

```
Interval<1> I(10);                          // I = {0, 1, ..., 9}
Array<1,double,Brick> A(I);
Range<1> J(0,8,2);                          // J = {0, 2, ..., 8}
Array<1,double,BrickView> B = A(J);
```

The new array `B` is a view of the even elements of `A`: {`A(0), A(2), ..., A(8)`}. Note that views always act as references (i.e., `B(0)` is an alias for `A(0)`, `B(1)` is an alias for `A(2)`, etc.). The task of determining the type of view engine to use when subsetting an `Array` is handled by the `NewEngine` traits class. Specializations of

the class template `NewEngine` define a trait `Type_t` that specifies the type of engine that is created when a particular engine type is subsetted by a particular domain type. Thus, in the above example we could have written:

```
typedef
  NewEngine< Engine<1,double,Brick>, Range<1> >::Type_t View_t;
Array<1,double,View_t::Tag_t> B = A(J);
```

While users can explicitly declare view-engine-based array objects in the manner above, these views will usually be created as temporaries via subscripting and then used in expressions and function calls to specify the elements on which to operate. For example:

```
Interval<1> I(10), I2(2,5);                  // I2 = {2, 3, 4, 5}
Array<1,double,Brick> A(I), C(I);
C(I2) = A(I2+1) - A(I2-1);            // C(2) = A(3) - A(1), etc.
```

The final expression builds three temporary views and then executes the expression on these views.

In multidimensional cases, there can be multiple interpretations of certain expressions involving views of arrays. For example, if `I` and `J` are domain objects, then what does `A(I,J) = B(J,I)` mean? If `I` and `J` are `Interval` objects of equal length, then this would be an element-wise assignment. However, POOMA II's `Index<N>` domain objects have knowledge of their loop ordering. If these domain objects are used, then `A(I,J) = B(J,I)` assigns the transpose of `B` to `A`. Thus, the user can choose between tensor-like subscript semantics and Fortran 90 array semantics simply by choosing different domain types.

4 Expressions and Evaluators

Most of the computation in a POOMA II code takes place in mathematical expressions involving several arrays. Expression templates and template metaprograms [3] are used to support an expressive syntax and to implement a number of compile-time optimizations. The most common of these optimizations is converting these high-level expressions into efficient low-level loops.

POOMA II maintains abstraction barriers between expression manipulation, evaluation and data storage. This allows POOMA II to generate efficient code for all types of engines, regardless of whether the data is stored locally or in a distributed fashion.

Expression templates work by storing the parse tree of an expression with operator objects at non-leaf nodes and data objects at the leaves. An expression object is templated on a type that encodes the structure of the parse tree so that the parse tree can be manipulated at compile time to produce efficient code. Consider the sample expression parse tree shown in Fig. 1. PETE encodes this parse tree in an object of type

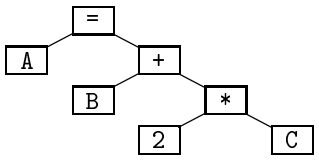

Fig. 1. Parse tree for the expression `A = B + 2 * C`

```
TBTree< OpAssign, Array1
 TBTree< OpPlus, ConstArray2,
  TBTree< OpMultiply, Scalar<int>, ConstArray3 > > >
```

containing references to arrays `A`, `B` and `C`, and the scalar 2. This expression object can be used to generate an optimized set of loops. However, it does not have array semantics and is not an `Array`, so it cannot be passed to functions expecting an `Array`.

The POOMA II engine architecture provides a solution to this problem: the *expression* engine. An expression engine wraps a PETE expression with an engine interface. Values of an expression engine are computed efficiently by the expression-template machinery based on the data referred to in an expression object. With this innovation, the result of an expression involving `Array` objects is an `Array`. Thus, users can write functions that operate on expressions by templating them for arbitrary engine types. For example,

```
template<int Dim, class T, class ET>
T trace(const ConstArray<Dim,T,ET> &a) {
  T tr = 0;
  for (int i = 0; i < a.length(0); ++i) { tr += a(i,i); }
  return tr;
}
```

Then `trace(B+2*C)` sums the diagonal components of `B+2*C` without computing any of the off-diagonal values.

Expression evaluation is a separate component from the array and expression objects. Evaluators only require a few basic services from arrays and expressions: subsetting, returning an element, getting a domain, etc. Any object that can use those services to evaluate expressions qualifies as an *Evaluator*. Expression evaluation is triggered by the assignment operator of `Array`, which builds a new `Array` that has an expression engine and hands it off to an `Evaluator`. Each expression is defined on a domain, and the `Evaluator` invokes a function specialized on the type of the domain to evaluate the expression at each point.

For example, suppose an expression is defined on a domain that has only STL-style iterators for looping over the domain. Then, if the domain object is `dom` and the expression-array object is `expr`, the inner evaluation loop could look like

```
for (dom::iterator dp = dom.begin(); dp != dom.end(); ++dp)
  expr(*dp);
```

If the domain is a two-dimensional `Interval`, for which we know that the strides are all unity, the inner loops would look like

```
for (int j = 0; j < dom[1].length(); ++j)
  for (int i = 0; i < dom[0].length(); ++i)
    expr(i, j);
```

The type of inner loop can be determined at compile time since it depends on the type of the domain. That allows the most specialized—and therefore the most efficient—code to be used for the provided data structures.

The `Evaluator` classes also provide a where-block interface, enabling code such as

```
where(A < 1);
  B = A;
elsewhere();
  B = 1 - A;
endwhere();
```

This code sets array `B` to `A` wherever `A` is less than 1 and `1 - A` otherwise. Each call to `where()`, `elsewhere()` and `endwhere()` manipulates state information in the evaluator that influences how expressions are evaluated.

One way to store this state is as a boolean mask array. Because where-blocks can be nested, there must be a stack of such masks, and the top of the stack is the mask for the currently active where-block. Alternatively, one can store a one-dimensional vector of discrete points where the expression is to be evaluated. This would be more efficient than the boolean mask if a small fraction of the mask is true. In either case, the `Evaluator` extracts an evaluation domain from the where-block expression and evaluates the expression at each point.

The evaluator system is designed to be extensible. Key extensions that are now under development include:

1. **Multiblock.** Multiblock arrays decompose their data into multiple blocks. The evaluator intersects the subdomains of a multiblock expression, subsets the expression with the intersections, and then evaluates the expression on each subdomain.
2. **SPMD Parallel.** In an SPMD parallel environment, the evaluator employs an algorithm such as owner-computes to decide what part of the whole domain should be evaluated on the local processor. It then takes a local view of the expression on that domain. If arrays in the expression have remote data, they must transfer their remote data in order to provide a local view. Once this view is constructed, it can be evaluated efficiently.
3. **Advanced Optimizations.** When an expression is ready for evaluation, it need not be evaluated immediately so long as there is a mechanism to account for data dependencies. There are two important reasons for deferred evaluation:
   (a) *Cache optimization.* A given calculation often involves a series of statements that use particular arrays multiple times, but each array is too large to fit in cache. In that case, it is more efficient to block each statement and evaluate one block for a series of statements before working on the next block.
   (b) *Overlapping communication and computation.* Typically the parts of a statement that require communication are along the boundaries of the domain for a given processor. Computation in the interior can proceed while communication needed for the boundaries is taking place.

5 Performance

In order to illustrate performance characteristics of POOMA II, we present a sample results using a stencil benchmark code. A stencil expression is an array expression that involves the same array object evaluated at several nearby points. Such expressions occur frequently in the numerical approximation of partial differential equations, and thus it is important that such expressions be evaluated efficiently.

Consider the simple stencil expression

```
B(I) = K * (A(I+1) - A(I-1))
```

To produce optimal code, the compiler must know that `B` is not aliased to anything on the right-hand side. It can put `&B`, `&A` and `K` in registers and unroll the loop so that it can save `A(I+1)` in a register and reuse this value when it needs `A(I-1)`. Accomplishing these optimizations is non-trivial. First, the compiler must be told, via the `restrict` keyword, that `B` is not aliased. Second, the compiler must be able to see that both occurrences of `A` refer to the same array. This is not guaranteed with expression templates, since the pointer to the array being operated on is buried in a `TBTree` node. Failure to realize this will not only prevent loop unrolling, but also result in the use of extra registers. For large stencils, the compiler may run out of registers ("register spillage"), which greatly impacts performance [6].

These problems can be overcome by encapsulating the stencil operation in a class. A stencil object calculates the value of the stencil given an array and an index. POOMA II stencil objects are fully integrated with the expression-template machinery.

Our stencil benchmark compares four approaches to the evaluation of the 9-point stencil

```
B(I,J) = c * ( A(I-1,J-1) + A(I,J-1) + A(I+1,J-1) +
               A(I-1,J)   + A(I,J)   + A(I+1,J)   +
               A(I-1,J+1) + A(I,J+1) + A(I+1,J+1) );
```

The evaluation methods are C code with `restrict`, C-style code using POOMA II arrays (C++ indexing), POOMA II code using expression templates (POOMA II Unoptimized), and POOMA II using stencil objects (POOMA II).

The benchmark was performed on an SGI Origin 2000 with 32 KB of primary cache, 4 MB of secondary cache, and a theoretical peak performance of 400 MFlops. Figure 2 shows the results for the four evaluation techniques using $N \times N$ arrays, where N ranges from 10 to 1000. The C code runs significantly faster than the all the C++ versions because it exploits the `restrict` keyword. For $N > 40$, the arrays are larger than primary cache, but there is little effect on performance. For $N > 400$, the arrays are larger than secondary cache, which leads to a large speed reduction. As the curves for the POOMA II unoptimized and stencil-object versions demonstrate, there is non-zero overhead in the expression-template machinery for small N. The advantage of the stencil-object

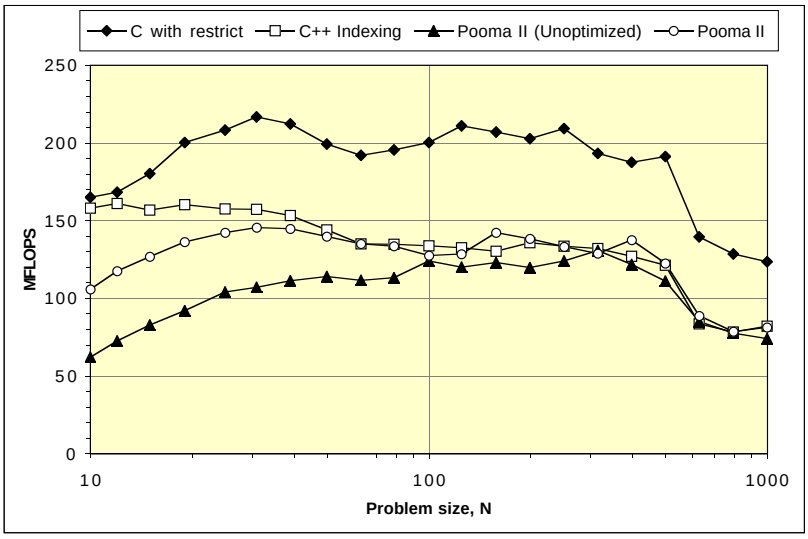

Fig. 2. Stencil benchmark results

approach over the unoptimized approach is clearly visible for $N < 100$. This does not persist for large N because the loop is not unrolled (no `restrict`) and the stencil is not sufficiently large to cause register spillage. An important result is that the stencil-object version performs almost identically to the C++ indexing version for $N > 30$. Once `restrict` is fully supported for C++, the performance of stencils implemented using POOMA II should closely approach that of C.

References

1. William Humphrey, Steve Karmesin, Federico Bassetti, and John Reynders. Optimization of data-parallel field expressions in the POOMA framework. In *ISCOPE '97*, December 1997. Marina del Rey, CA.
2. John Reynders et al. POOMA: A framework for scientific simulations on parallel architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.
3. Todd Veldhuizen. Expression templates. *C++ Report*, June 1995.
4. James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.
5. David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.
6. Federico Bassetti, Kei Davis, and Dan Quinlan. A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks. In *ISCOPE '97*, December 1997. Marina del Rey, CA.